

Block-Level Parallelism in Parsing Block Structured Languages

Abhinav Jangda

Indian Institute of Technology (BHU), Varanasi
abhinav.student.apm11@iitbhu.ac.in

Abstract. Softwares source code is becoming large and complex. Compilation of large base code is a time consuming process. Parallel compilation of code will help in reducing the time complexity. Parsing is one of the phases in compiler in which significant amount of time of compilation is spent. Techniques have already been developed to extract the parallelism available in parser. Current LR(k) parallel parsing techniques either face difficulty in creating Abstract Syntax Tree or requires modification in the grammar or are specific to less expressive grammars. Most of the programming languages like C, ALGOL are block-structured, and in most languages grammars the grammar of different blocks is independent, allowing different blocks to be parsed in parallel. We are proposing a block level parallel parser derived from Incremental Jump Shift Reduce Parser by [13]. Block Parallelized Parser (BPP) can even work as a block parallel incremental parser. We define a set of Incremental Categories and create the partitions of a grammar based on a rule. When parser reaches the start of the block symbol it will check whether the current block is related to any incremental category. If block parallel parser find the incremental category for it, parser will parse the block in parallel. Block parallel parser is developed for LR(1) grammar. Without making major changes in Shift Reduce (SR) LR(1) parsing algorithm, block parallel parser can create an Abstract Syntax tree easily. We believe this parser can be easily extended to LR (k) grammars and also be converted to an LALR (1) parser. We implemented BPP and SR LR(1) parsing algorithm for C Programming Language. We evaluated performance of both techniques by parsing 10 random files from Linux Kernel source. BPP showed 28% and 52% improvement in the case of including header files and excluding header files respectively.

1 Introduction

Multi core chip architectures are emerging as feasible solution to effectively utilizing the ever growing number of chip. Multi-core chip depends on success in system software technology (compiler and runtime system), in order to have thread level parallelism and utilizing on-chip concurrency. With multi-core processors, additional speedups can be achieved by the use of parallelism in data-independent tasks. There is a gradual shift towards making current algorithms and design into parallel algorithms. It is rather difficult to achieve lock free and

low cache contention parallel algorithms.

In 70s papers appeared ideas on parallel compilation of programming languages and parallel execution of programs were expected. In those papers discussions on parallel lexical analysis, syntactic analysis and code generation were done. With VLSI applications, prominent increase in research on parallel compilation is observed.

A compiler contains different phases: lexical analyzer, syntactic analyzer, semantic analyzer and code generator. Parsing or syntax analysis is the phase of compiler which analyses the program code according to the language. After analysis, it converts the code into another formal representation which will act as input for succeeding phases of compiler.

Complexity of software source code is increasing. An effort to compile large code base is very time consumable. [4] describes two types of parsers: Top Down and Bottom Up Parsers. Top Down parsers have less power as compared to Bottom Up Parser. LR (k), SLR (k) and LALR (1) are types of Bottom Up Parsers. With more power Bottom Up Parsers also requires more space and more time in parsing a string as compared to Top Down parsers. Most of the compiler compilers like Yacc [17] and Bison [16] creates LR (1) parsers and compilers like clang [18], Mono C# Compiler [19] etc. uses LR(1) parsers. So, it is evident that programming languages can be represented easily by LR (1) languages.

Parsing is very time consuming phase of compiler. Parsing different files in parallel is not enough. As programming languages like C and C++ can includes different files (using #include) in a single file which results in generation of very long file. If we can parallel the parsing phase of single file, it will give performance benefits in compiling the source code. Many significant techniques are already proposed for making parallel parsers ([2], [8], [10], [9], [11]). A parallel parsing for programming language is given by [14].

A block is a section of code which is grouped together. In a language, a block may contain class definition, member or method declaration. Another block could be a block of statements also called compound statement. This block is usually associated with a function code or if statement or loop. Programming Languages such as C, C++, Java, Python use the concept of blocks heavily. One of most important property of parsing blocks is that they all are independent of each other i.e. each block can be parsed independently of other block. So, we could parse many blocks in a parallel fashion. In this paper, we will propose a technique to parse various blocks of the code in parallel. It can also work as a block parallel incremental parser. Our parser is termed as Block Parallelized Parser (BPP, for short).

Our technique of parallel parsing is based on incremental parsing. An incremental parser is the one that parse only those portions of a program that have been modified. Whereas an ordinary parser must process the entire program when it is modified. An incremental parser takes only the known set of changes done in a source file and updates its internal representation of source file which may be an Abstract Syntax Tree. By building upon the previously parsed files, the incremental parser avoids the wasteful re-parsing of entire source file where

most of the code remains unchanged.

BPP is based on the properties that an incremental parser can parse any part of a source code without the need of parsing the whole source code and different blocks in a source code can be parsed independently of other blocks. In BPP these parts are blocks in a source code. Using the property of incremental parser, BPP parse each of the blocks independently of other blocks. Each of these blocks are parsed in their own thread. It can be easily seen that BPP follows a divide and conquer approach. It divides the source into different blocks, parse each of them in parallel and at the end conquer these blocks. In our scheme the conquer step does nothing except waiting for all the BPP Threads to complete their operations.

There have been many works on incremental parsing [Incremental Parsing References]. We choose to extend on the works of Incremental Jump Shift Reduce parser of [13]. BPP is derived from Incremental Jump Shift Reduce Parser. In [13], authors defined Incremental Jump Shift reduce parser for SLR (1) languages only. However, we decided to extend this parser to accept LR(1) language because LR(1) languages has more power than SLR (1) and nearly all programming languages can be defined in the form of LR(1) grammars. We define the incremental categories to be a statement containing a block like class definition or function definition or if statement or for loop statement. Then, we give a notion of First Non-Terminal symbols of a Non-Terminal symbol. We used this notion to create partitions of a grammar such that a partition includes an incremental category and its First Non-Terminals. We observed that this scheme gives us a very interesting property in Incremental Jump Shift Reduce parser. We used this property to create our Block Parallelized Parser.

Whenever a start of the block symbol is encountered the parser will first check whether the current block is related to any incremental category and can it be parsed independently. If BPP is able to find the incremental category for it, BPP will start parsing the block in parallel. In this paper we developed this BPP for LR(1) languages but we believe it can be easily extended to LR(k) or can be easily converted to LALR (1) or SLR (1) grammars. We also see that no major changes were done to the current LR(1) parsing algorithm and hence, it should be easy to create an Abstract Syntax Tree. This parser can also work as an incremental parallel parser which can parse different blocks in parallel. Moreover, it could be seen that there is no requirement of any Thread Synchronization to communicate between different threads of BPP each of which is parsing a block in parallel. This is because no two blocks are related in any way for the purpose of parsing.

We compiled C# implementation using Mono C# Compiler 3.12.1 and executed the implementation using Mono JIT Compiler 3.12.1 on machine running Fedora 21 with Linux Kernel 3.19.3 with 6 GB RAM and Intel Core i7-3610 CPU with HyperThreading enabled. We found out that our technique showed 28% performance improvement in the case of including header files and 52% performance improvement in the case of excluding header files.

The following paper is designed as follows. Section 2 shows some previous

work done in parallel parsing. Section 3 and 4 provides the terminology we will use and the background required to understand our technique. In Section 5 we will extend Incremental Jump Shift Reduce parser to accept LR(1) grammars. In Section 6, we introduced the concept of First Non Terminals of a non terminal. In Section 7 we will use this concept to create partitions of the grammar. We also showed that by creating partitions using this concept we get a very interesting property. This property would be used by BPP. We have generalized this property in a theorem and also provided a proof for it. In Section 8 we presents our Block Parallelized Parser and its parsing algorithm. In Section 9 we will compare our algorithm with previous work. Section 10 shows our evaluation and results. In Section 11 and Section 12 we complete our document with conclusion and related work.

2 Related Work

A lot of previous work has been done in Parallel Parsing of LR (1) and Context Free Languages. The most recent work done by [2] in parallel parsing of LR(1) is an extension of an LR substring parser for Bounded Context Languages (developed by Cormack) for Parallel environment. [3] provided a substring parser for Bounded Context-LR Grammars and Simple Bounded Context-LR Grammars. [2] distributes the work of parsing the substrings of a language over different processors. The work was extended to different processors in a Balanced Binary Tree Fashion and achieved $O(\log n)$ time complexity of parsing. But constructing a Bounded Context LR Grammar for a programming language is also difficult. C++ is one of the programming languages which cannot be parsed by LR (1) parsing [12] so creating a Bounded Context Grammar is out of question here.

Parallel and distributed compilation schemes can be divided into two broad categories, functional decomposition and data decomposition. [1] and [20] talks about distributed compilation using a scheme based on functional decomposition. Functional decomposition scheme divides different phases of compiler: lexer, parser, semantic analyzer into functional component and running each of them on separate processors like an instruction pipeline fashion. The data decomposition scheme divide the input into sections of equal length and parse them in parallel. BPP is data decomposition scheme which parallel the parser by divide and conquer approach. The data decomposition scheme was developed by [8], [7], [9], [11]. These schemes are parsing LR (k) in parallel. They divide the input into sections of equal length and then parse them in parallel. [10], [11], [8] describes asynchronous algorithms while [9] develops a synchronous algorithm. [11] develops a parallel LR parser algorithm using the error recovery algorithm of [21].

[1] has developed an Incremental Parallel Compiler which could be used in Interactive Programming Environment and he developed an Incremental Parallel Parser also. [22] improves upon the Divide and Conquer Parsing technique developed by [23]. They show that while the conquer step of algorithm in [23] is $O(n^3)$ but under certain conditions it improves to $O(\log^3 n)$.

[14] describes a grammar partitioning scheme which would help in parsing the language in parallel. In [14] a type of Statement Level Parallelism has been developed. The grammar is divided into n different sub-grammars corresponding to n subsets of the language which will be handled by each sub-compiler. For each n sub grammars required to generate parse tables (using parser generator) along with driver routine constitute a parser for sub-compiler. For each sub-compiler, a requirement of modified scanner is there which recognizes subset of the language. The technique described in [14] requires a lot of modification to the Lexical Analyzer. A separate Lexical Analyzer has to be developed for one type of language. The parser of [14] requires automatic tools for its implementation.

In all the above described techniques constructing Abstract Syntax Tree for a Block Structured Language is difficult. As Blocks in a Block Structured Language are independent on each other, so they can be parsed independently. Moreover this scheme would not involve Inter-Thread Communication before a thread completes its Parsing of Blocks. Hence, no shared memory synchronization methods are required to coordinate between different threads. It could be easily seen that the creation of an Abstract Syntax Tree is also very easy. With all these required things in mind we have developed Block Parallelized Parser for LR (1) languages.

3 Terminology

We assume the notation for Context Free Grammar is represented by $G = (N, T, S, P)$ where N is set of non-terminals, T is set of terminals, S is start symbol and P is set of productions of the grammar. The language generated by G is given as

$$L(G) = \left\{ \omega \in T^* \mid S \xRightarrow{*} \omega \right\}$$

We will use the following conventions.

$$\begin{aligned} S, A, B, \dots &\in N \\ a, b, \dots &\in T \\ \dots, w, x &\in T^* \\ X, Y &\in N \cup T \\ \alpha, \beta, \gamma, \dots &\in (N \cup T)^* \end{aligned}$$

Given a grammar G , we represent its augmented grammar as $G' = (N', T', S', P')$, where

$$\begin{aligned} N' &= N \cup \{S'\} \\ T' &= T \cup \{\$ \} \\ P' &= P \cup \{S' \rightarrow S\$ \} \end{aligned}$$

Here S' is called the augmented start symbol of G' and $\$$ is the end of string marker. We denote a set of End Of String markers as EOS.

In our paper we will represent Block Parallelized Parser as BPP, Jump Shift Reduce parser as JSR and Incremental Jump Shift Reduce parser as IJSR. An LR(1) item is represented as $[A \rightarrow \alpha.\beta, a]$, where a is the lookahead symbol.

In a programming language, a block represents a section of code grouped together. This section of code can be a group of statements, or a group of declaration statements. For example in Java, a block corresponding to class defini-

```

/* The Top Level Block */
import java.util.*;
import java.awt.*;

class MessageBox
{
    /* First Level Child Block */
    class OkButton
    {
        /* Second Level Child Block */
    }
    void show()
    {
        /* Second Level Child Block */
    }
}

```

Fig. 1. Top-Level and Child Blocks of a Java Program.

tion contains declaration statements for fields and methods. Block corresponding to function definition can contain declaration statement for local variables or expression statements or control flow statements. A Top-Level Block is the starting block of a program which contains definition for Classes, Functions, Import/Include statements etc. Child Blocks are contained in either Top-Level Block or another Child Block. As we proceed further, Block will be in reference to Child Block. Fig. 1, shows an example of Top-Level and Child Blocks of a Java Program.

A start block symbol could be " $\{$ " in C style languages, "begin" in Pascal style languages is represented as terminal s_b . An end block symbol which could be " $\}$ " in C style languages or "end" in Pascal style languages is represented as e_b .

4 Background

We now survey LR (1) parsers and their generation algorithm as given by [4]. LR (1) parsers are table driven Shift Reduce parsers. In LR (1), L denotes left-to-right scanning of input symbols and R denotes constructing right most derivation in reverse. Some extra information is indicated with each item: the set of possible terminals which could follow the items LHS. This set of items is called the lookahead set for the item. Here, 1 signifies that number of lookahead symbols required are 1.

LR (1) parser consists of an input, an output, a stack, a driver routine. A driver routine runs its parsing algorithm which interacts with two tables ACTION and GOTO. Any entry in ACTION and GOTO tables are indexed by a

symbol which belongs to $N \cup T'$ and the current state. An entry in both the tables can be any one of the following:

- If ACTION $[j, a] = \langle S, q \rangle$ then a Shift Action must be taken.
- If ACTION $[j, a] = \langle R, A \rightarrow \alpha \rangle$ then reduce symbols to a production.
- If ACTION $[j, a] = \text{Accept}$ then grammar is accepted.
- If ACTION $[j, a] = \text{error}$ then Error has occurred.
- If GOTO $[j, A] = q$ then go to state q .

An LR (1) item is of the form $[A \rightarrow \alpha.\beta, a]$, where a is a lookahead symbol. Construction of LR (1) items requires two procedures CLOSURE and GOTO. CLOSURE takes a set of items as its argument. GOTO takes a set of items and a symbol as arguments. Both are defined as follows:

$$\text{CLOSURE}(I) = I \cup \{[B \rightarrow \gamma, b] \mid [A \rightarrow \alpha.B\beta, a] \in I \text{ and } B \rightarrow \gamma \in P' \forall b \in \text{FIRST}(\beta a)\}$$

$$\text{GOTO}(I, X) = \text{CLOSURE}(\{[A \rightarrow \alpha X.\beta, a] \mid [A \rightarrow \alpha.X\beta, a] \in I\})$$

Collection of set of LR (1) items is created using CLOSURE and GOTO functions. Items function creates the collection of set of LR (1) items.

$$\text{items}(G') = \text{CLOSURE}(\{[S' \rightarrow .S, \$]\}) \cup \{\text{GOTO}(I, X) \mid I \in C \text{ and } X \in P'\}$$

ACTION and GOTO tables are created using this collection. Following is the procedure to create these tables:

1. Create collection of set of LR(1) items. Let this collection be $C' = \{I_0, I_1, I_2, \dots, I_n\}$
2. Let i be the state of a parser constructed from I_i . Entries in ACTION table are computed as follows:
 - (a) ACTION $[i, a] = \text{shift } j$, if $[A \rightarrow \alpha.a\beta, b] \in I_i$ and $\text{GOTO}(I_i, a) = I_j$
 - (b) ACTION $[i, a] = \text{reduce } A \rightarrow \alpha.$, if $[A \rightarrow \alpha., a] \in I_i$ and $A \neq S'$
 - (c) ACTION $[i, \$] = \text{accept}$, if $[S' \rightarrow S., \$] \in I_i$
 - (d) GOTO $[i, A] = j$, if $\text{GOTO}(I_i, A) = I_j$
3. All other entries not defined by (b) and (c) are set to error.
4. The Initial State is the one containing the item $[S' \rightarrow .S, \$]$.

Most of the programming languages could be constructed from LR (1) grammar. Hence, LR (1) is the most widely used parser. Many parser generators like YACC and GNU Bison generates an LR (1) parser.

A Jump Shift Reduce [13] (JSR in short) parser is an extension of LR (1) parser. LR (1) parser generates ACTION and GOTO table for the augmented grammar G' . JSR parser first partition the grammar G' into several sub grammars and creates parsing sub- table of every sub grammar. Hence, the ACTION and GOTO tables of LR (1) are split into several ACTION and GOTO tables in JSR parser. JSR parser is equivalent to the LR (1) parser that is it will only accept languages generated by LR (1) grammar [13].

Let $G' = (N', T', S', P')$ be the augmented grammar of grammar $G = (N, T, S, P)$. Let us partition the grammar G on the basis of Non Terminals. Let G^i denotes a partition of the grammar G , such that we have

$$G^i = (M^i, T^i, S^i, P^i)$$

where,

$$N^i \subseteq N \text{ such that } N^i \cap N^j = \emptyset \mid i, j = 1, \dots, n, i \neq j$$

$$\begin{aligned}
P^i &= \{A \rightarrow \alpha \in P \mid A \in N^i\} \forall i = 1, \dots, n \\
M^i &= N^i \cup \{B \in N \mid \exists A \rightarrow \alpha B \beta \in P^i\} \\
T^i &= \{a \in T^i \mid \exists A \rightarrow \alpha a \beta \in P^i\} \\
S^i &\in N^i
\end{aligned}$$

Therefore, we have

$$(\cup N^i, \cup T^i, S', \cup P^i) = G'$$

For every subgrammar G^i , a parsing subtable named $\text{Tab}(S^i)$ is built. Each subtable contains ACTION and GOTO subtables which are represented by $\text{Tab}(S^i).\text{ACTION}$ and $\text{Tab}(S^i).\text{GOTO}$. In addition to the Shift, Reduce and Accept action there is an additional Jump action. Jump action is associated with a sub-table. Whenever a Jump action is encountered then the parsing algorithm Jumps to a sub-table and parse the partition related to that sub-table.

We will now investigate few points about the Incremental Jump Shift Reduce Parser [13]. Incremental Jump Shift Reduce Parser (IJSR) [13] is based upon the JSR parser [13]. A set of Incremental Categories will be defined which could be incrementally parsed by IJSR parser. Given a grammar $G = (N, T, S, P)$ a set of Incremental Categories has to be defined $IC = \{C_j \mid C_j \in N, j = 1, 2, \dots, n\}$ and the Incremental Language of G is

$$L^*(G) = \cup L^*(A),$$

where $A \in IC \cup \{S\}$ and $L^*(A) = \{\alpha \in (T \cup IC)^* \mid A \xRightarrow{*} \alpha\}$

For every Incremental Category A , add a production $A \rightarrow A\#_A$, where $\#_A$ is an end-of-string for A . For a given grammar $G = (N, T, S, P)$ with a set of Incremental Categories $IC = \{C_j \mid C_j \in N, j = 1, 2, \dots, n\}$ an Incremental Grammar is defined as

$$G^* = (N, T \cup EOS, S, P \cup P_{IC}),$$

where, EOS is the set of End of String markers = $\{\#_j \mid j = 1, 2, \dots, n\}$
 $P_{IC} = \{C_j \rightarrow C_j\#_j \mid C_j \in IC, \#_j \in EOS\}$

A major change by this extension is that now the strings may contain incremental symbols which are also non-terminal symbols. The difference between ACTION and GOTO tables disappears, as an incremental category can also occur in the input string and can be shifted on the stack. Hence, we would have only ACTION table and no need for GOTO table. As we have also introduced EOS set, the ACTION table can now be indexed with symbols belonging to $N \cup T' \cup EOS$. Every Incremental Category will have its own start state and accept action. We will represent the subtable as $\text{Tab}(S^t)$. Entries of table will be as follows:

- If $\text{Tab}(S^t)[j, X] = \langle S, q \rangle$ then a Shift Action must be taken.
- If $\text{Tab}(S^t)[j, X] = \langle R, A \rightarrow \alpha \rangle$ then a Reduce Action must be taken.
- If $\text{Tab}(S^t)[j, \$] = \text{Accept}$ then input is accepted.
- If $\text{Tab}(S^t)[j, \#_i] = \text{Accept}$ then input for Incremental Category C_i will be accepted.
- If $\text{Tab}(S^t)[j, a] = \langle J, K \rangle$ then jump to a subtable $\text{Tab}(S^k)$.
- If $\text{Tab}(S^t)[j, a] = \text{error}$ then error occurred.

5 Extending LJSR Parser to accept languages generated by LR (1) grammars

As JSR Generation Algorithm was already developed for LR (0) items and Incremental JSR Generation Algorithm was developed for SLR (0) items [13]. In this section, we will first extend the JSR Parser Generation Algorithm to accept LR (1) Grammar and then we will extend LJSR parser to accept LR(1) Grammar.

Generation of subtables first requires the generation of canonical collection of sets of augmented items. An augmented item is a triplet represented as $\langle i, FF, TF \rangle$, where i is an LR item, FF called From-Field and TF called To-Field are the names of parsing sub-tables. From-Field represents the sub-table which contains the last action performed by parser and To-Field represent the sub-table which contains the next action to be performed. Although we focus only LR (1) items but the procedure for LR (k) items is very similar.

To-Field of an augmented item of a state is determined using TO function. Let us define the TO function. Function TO calls Function CHOOSE_NEXT.

```

1: procedure TO( $I_j$ )
2:    $I_j'' = \phi$ 
3:   for all item  $i_k$  in  $I_j$  do
4:     if  $i_k$  is  $[A \rightarrow \alpha.a\beta, b]$  then
5:        $\text{add } \langle i_k, \text{CHOOSE\_NEXT}(I_j, a) \rangle$  to  $I_j''$ 
6:     else if  $i_k$  is  $[A \rightarrow \alpha.B\beta, b]$  then
7:        $\text{add } \langle i_k, S^i \rangle$  to  $I_j''$ , where  $B \in N^i$ 
8:     else if  $i_k$  is  $[A \rightarrow \beta., b]$  then
9:        $\text{add } \langle i_k, S^i \rangle$  to  $I_j''$ , where  $A \in N^i$ 
10:  return  $I_j''$ 

```

This function selects a parsing table out of those in which the parsing of the remaining string could continue. Let $< \cdot$ be a total ordering relation over the set $ST = \{S^p \mid p = 1, 2, \dots, n\}$ of the names of parsing sub-tables, such that $S^i < \cdot S^{i+1}$, $i = 1, 2, \dots, n-1$

From-Field of an augmented item is enriched using FROM function. FROM

```

1: procedure CHOOSE_NEXT( $I_j, a$ )
2:   let  $ST'$  be a set of parsing subtables
3:   for all item  $[H \rightarrow \alpha.a\beta, b]$  in  $I_j$  do
4:      $\text{add } S_h$  to  $ST'$  such that  $H \in N_h$ 
5:   return  $\min < \cdot ST'$ 

```

takes two arguments I_t'' , which is a set of items enriched with To-Field and I_j , whose items we have to enrich with From-Field.

STATES procedure is used to generate the collection of set of JSR items.

```

1: procedure FROM( $I_t'', I_j$ )
2:    $I_j' = \phi$ 
3:   for all  $item i_k \in I_j$  do
4:     if  $i_k$  is  $[S' \rightarrow .S\$, b]$  then
5:        $add < i_k, S^1 > to I_j'$ 
6:     else if  $i_k$  is  $[A \rightarrow \alpha X.\beta, b]$  then
7:        $add < i_k, TF > to I_j', where < [A \rightarrow \alpha X.\beta, b], TF > \in I_t''$ 
8:     else if  $i_k$  is  $[A \rightarrow .\beta, b]$  then
9:        $add < i_k, FF > to I_j', where < [B \rightarrow \alpha.A\beta, b], FF > \in I_j'$ 
10:  return  $I_j'$ 

```

STATES algorithm first generates the collection of sets of LR(1) items using ITEMS procedure which was discussed previously. Afterwards, it calls TO and FROM functions to generate set of augmented items from the corresponding set of LR(1) items.

We will now extend LJSR parser to accept LR (1) languages. This extended

```

1: procedure STATES( $G'$ )
2:    $C' = items(G')$ 
3:    $I^A = \phi$ 
4:   for all  $I_j \in C'$  do
5:      $I_j' = \{ < i_k, FF_k > \mid i_k \in I_j \} = FROM(I_t'', I_j), where I_j =$   

 $GOTO(I_t, X) \text{ and } I_t = \phi \text{ if } j = 0$ 
6:      $I_j'' = \{ < i_k, TF_k > \mid i_k \in I_j \} = TO(I_j)$ 
7:      $I_j^A = \{ < i_k, FF_k, TF_k > \mid < i_k, FF_k > \in I_j' \text{ and } < i_k, TF_k > \in I_j'' \}$ 
8:      $I^A = I^A \cup \{ I_j^A \}$ 
9:  return  $I^A$ 

```

parser is based on the previous JSR parsing algorithm. The FIRST function used to compute the set of First symbols related to a non-terminal has to be modified to include the incremental categories also. The reason being an incremental category can also occur in the input string, can be shifted on the stack while parsing and can reduce to a production. Moreover, FIRST should now also include the EOS markers. Hence, the new FIRST becomes

$$FIRST(A) = \{a \mid A \xRightarrow{*} a\beta, \text{ where } a \in T' \cup EOS \cup IC\}$$

For an incremental category A , there will be a set of items containing item $[A \rightarrow .A\#_A, \#_A]$ and items $[B \rightarrow \alpha.A\beta\#_A]$. Then the state corresponding to this set of items will be the start state of the incremental grammar corresponding to A . Correspondingly, there will a single set of items that contains the item $[A \rightarrow A.\#_A, \#_A]$. The state belonging to this set of item is the accepting state of the incremental grammar corresponding to A . The LJSR parser has initial and final states for every incremental category.

Now, we can extend the LJSR parser for accepting languages generated by

LR(1) grammars. The procedure L-TABS given below is used to construct the LJSR parsing table.

```

1: procedure L-TABS( $G'$ )
2:    $C' = \text{items}(G')$ 
3:   for all  $I_j^A \in C'$  do
4:     for all  $\langle i_h, H, K \rangle \in I_j^A$  do
5:       if  $i_h$  is of the form  $[A \rightarrow \alpha.X\beta, b]$  then
6:          $\text{Tab}(S^K)[j, X] = \langle S, q \rangle$  where  $\text{GOTO}(I_j, X) = I_q$ 
7:         if  $H \neq K$  then
8:            $\text{Tab}(S^H)[j, X] = \langle J, K \rangle$ 
9:       else if  $i_h$  is of the form  $[A \rightarrow \alpha., X]$  then
10:         $\text{Tab}(S^K)[j, X] = \langle R, A \rightarrow \alpha \rangle$ 
11:        if  $H \neq K$  then
12:           $\text{Tab}(S^H)[j, X] = \langle J, K \rangle$ 
13:       else if  $i_h$  is of the form  $[S' \rightarrow S.\$, \$]$  then
14:         $\text{Tab}(S^{S'})[j, \$] = \text{accept}$ 
15:       else if  $i_h$  is of the form  $[A \rightarrow A.\#_A, \#_A]$  then
16:         $\text{Tab}(S^K)[j, \#_A] = \text{accept}$ 

```

6 First Non Terminals

In this section we will define the concept of First Non Terminals.

We define a set of First Non Terminals for a non terminal A as a set of non-terminals that appear at the beginning of any sentential form derived from A i.e. a set of non terminals B such that there exists a derivation of the form $A \xRightarrow{*} B\beta$. $FIRSTNT(A)$ represents the set of First Non Terminals for A and can be represented in set notations as:

$$FIRSTNT(A) = \bigcup_B \{B \mid A \xRightarrow{*} B\beta\}$$

To compute $FIRSTNT(A)$ for any non-terminal A , apply the following rules until no more terminals can be added to the $FIRSTNT(A)$ set.

1. If A is a terminal, then $FIRSTNT(A) = \phi$
2. If A is a non-terminal and $A \rightarrow B_1B_2...B_k$ is a production for some $k \leq 1$, then place B_i and everything in $FIRSTNT(B_i)$ in $FIRSTNT(A)$ if $B_1, B_2, ..., B_{i-1} \xRightarrow{*} \epsilon$
3. If $A \rightarrow \epsilon$ is a production, then $FIRSTNT(A) = \phi$

EXAMPLE 1: If we have following productions:

$S \rightarrow DAB$
 $S \rightarrow C$
 $A \rightarrow aB$

$B \rightarrow b$
 $C \rightarrow c$
 $D \rightarrow d$

Then find $FIRSTNT(S)$?

SOLUTION 1:

Due to first two productions of S we have,

$$\begin{aligned}
 FIRSTNT(S) &= FIRSTNT(A) \cup FIRSTNT(C) \cup \{A, C\} \\
 FIRSTNT(A) &= FIRSTNT(a) = \phi \\
 FIRSTNT(C) &= FIRSTNT(c) = \phi
 \end{aligned}$$

7 Using First Non Terminals to create partitions

In this section we will use the concept of First Non Terminals to create partitions of grammar. LJSR Parser will use these partitions to develop its tables. We will see that this kind of partitioning leads to a very interesting property in LJSR Parsing algorithm. We will generalize this property in a theorem and will also prove it.

We will partition the grammar in such way that:

- Every Incremental Category will have its own partition.
- The partition of Incremental Category will contain First Non Terminals of that incremental category also.
- Intersection of set of First Non-Terminals of any two incremental categories must be empty.
- All other remaining non-terminals including the augmented and start symbol are included in the first partition.

Given a grammar $G^* = (N, T \cup EOS, S, P \cup P_{IC})$ with a set of Incremental Category, $IC = \{C_t \mid C_t \in N\}$ we define partitions of non-terminals as N^2, N^3, \dots, N^n such that:

$$N^t = \{C_t\} \cup FIRSTNT(C_t) \text{ and } C_t \neq S$$

and

$$FIRSTNT(C_t) \cap FIRSTNT(C_s) = \phi, \text{ where } t, s = 1, 2, \dots, n \text{ and } t \neq s$$

And first partition,

$$N^1 = N - \bigcup_t N^t$$

Example 2: Partition the grammar given in Example 1 using first non terminals as given above and create LJSR parsing table. Then parse the string "dabb" and parse string "ab" incrementally for incremental category A.

Solution 2: First we have to create an augmented grammar of the given grammar by adding production $S' \rightarrow S\$$. Next we can create two partitions of grammar as given by the following partitions of non terminals.

$$\begin{aligned}
 N^1 &= \{S', S, B, C, D\} \\
 N^2 &= \{A\}
 \end{aligned}$$

As the incremental category we want is only A. Hence, there will be two

partitions, N^2 containing A and its first non terminals and N^1 will contain the remaining non terminals. Moreover, we would also have to add an EOS marker for A , let us say it is $\#_1$. We can generate the LJSR table using *L-TABS* procedure described in Section 5.

Let us parse the string "*dabb*". Table 1 shows the series of actions taken while parsing "*dabb*". In this case the start state will be the start state of table $Tab(S')$ i.e. 0. Table 2 shows the series of actions taken when "*ab*" is parsed incrementally with the incremental category A . In this case the start state will be the start state of table $Tab(A)$ i.e. 2.

Table 1. Parsing of "*dabb*"

STACK	INPUT	ACTION
0	dabb\$	Shift 5
0d5	abb\$	Reduce $D \rightarrow d$
0D2	abb\$	Jump A
0D2	abb\$	Shift 7
0D2a7	bb\$	Jump S'
0D2a7	bb\$	Shift 11
0D2a7b11	b\$	Reduce $B \rightarrow b$
0D2a7B10	b\$	Jump A
0D2a7B10	b\$	Reduce $A \rightarrow aB$
0D2A6	b\$	Shift 9
0D2A6b9	\$	Reduce $B \rightarrow b$
0D2A6B8	\$	Reduce $S \rightarrow DAB$
0S1	\$	Accept

Table 2. Parsing of "*ab*"

STACK	INPUT	ACTION
2	ab $\#_1$	Shift 7
2a7	b $\#_1$	Jump S'
2a7	b $\#_1$	Shift 11
2a7b11	$\#_1$	Reduce $B \rightarrow b$
2a7B10	$\#_1$	Jump A
2a7B10	$\#_1$	Reduce $A \rightarrow aB$
2A6	$\#_1$	Accept

In Table 1, when stack state reaches 0D2 there is a Jump to the Table $Tab(A)$. From this point until when stack state changes to 0D2A6, the actions taken are same as the actions of Table 2 and in the same table i.e. $Tab(A)$. Moreover, in between these Stack states in Table 1 "*ab*" is parsed to A .

We can generalize this example in the sense that same series of actions are taken when parsing a string and when parsing its substring incrementally for its incremental category. In the current example all the same actions happens in the same table because we created the partitions in such a way that all the first non terminals are in that partition. If the partitions were not created in the way described, it could have happened that these actions would happen in the different sub tables.

This technique is utilized by our BPP and it is generalized and proved in the theorem below.

Theorem 1 *Given an Incremental Grammar $G^* = (N, T, P', S')$ with set of incremental categories $IC = \{C_t \mid C_t \in N\}$ such that the non terminal partition, N^t related to incremental category C_t contains only C_t and $FIRSTNT(C_t)$. For*

an incremental category C_t and any terminal $b \in FIRST(C_t)$, if $C_t \xRightarrow{*} b\gamma$ and during parsing of the word $w = "\mu b\gamma\delta"$ the parser reaches at a state q in the subtable of C_t after performing the shift action on b then during the incremental parsing of the word $w_t = "b\gamma"$ for the incremental category C_t the parser will also reach the state q after performing the shift action on b in the sub table of C_t .

Proof. Outline: We will divide the proof in 4 cases. For each case, we will first find two sets of JSR items reached after performing shift on b one during the parsing of word w and other during the incremental parsing of word w_t related to incremental category C_t . We will then show that both of these sets contains same JSR items which implies the above theorem.

It is given that C_t is an incremental category and $C_t \neq S$ and let N^t be the partition containing only C_t and $FIRSTNT(C_t)$. Let S^t be the sub-table related to C_t . For any non-terminal $A \in FIRSTNT(C_t)$ and $A \rightarrow b\beta$, we must have $A \in N^t$.

During incremental parsing of the word $w_t = b\gamma$ for the incremental category C_t , the state before performing actions related to b will be the start state of the sub table $Tab(C_t)$. Let that start state be m .

We will have four cases on the basis of whether the grammar has productions of the form, $B \rightarrow cXC_t\beta$ and $B \rightarrow C_t\beta$

Case 1: If $B \rightarrow cXC_t\beta \in P'$ and $B \rightarrow C_t\beta \in P'$

Let the set of LR(1) items related to the start state of $Tab(C_t)$ i.e. state m be I_m and X lies in some partition other than N^t i.e. $X \in N^h$ and $t \neq h$.

As noted by [13] the start state of $Tab(C_t)$ must contain the item $[C_t \rightarrow .C_t\#_t, d]$, where d is a lookahead symbol. So we have $[C_t \rightarrow .C_t\#_t, d] \in I_m$. It is evident that the item $[C_t \rightarrow .C_t\#_t, d]$ should be result of a closure of another item. The only such item we can see is $[B \rightarrow cX.C_t\beta, j]$, where j is some lookahead symbol. So, we must have $[B \rightarrow cX.C_t\beta, j] \in I_m$. As discussed in Section 2, to get a set of LR(1) items we have to apply CLOSURE ($[B \rightarrow cX.C_t\beta, j]$). Hence, we have

$$\begin{aligned} [B \rightarrow cXC_t\beta, j] &\in I_m \\ [C_t \rightarrow .C_t\#_t, d] &\in I_m \\ [C_t \rightarrow .A\gamma\beta, e] &\in I_m \\ [A \rightarrow .b\beta, f] &\in I_m \end{aligned}$$

where e and f are some lookahead symbols.

Let I_o be the set of LR(1) items such that $I_m = GOTO(I_o, X)$. So, $[B \rightarrow cX.C_t\beta, j] \in I_o$. Let I_o'' be a set of JSR items enriched with TO fields corresponding to all LR(1) items in I_o . After applying TO procedure over I_o we would get the TO field for item $[B \rightarrow cX.C_t\beta, j]$ as S^h because $X \in N^h$.

To get TO fields for all JSR items corresponding to LR (1) items of the set I_m we have to apply TO procedure over I_m . We could see that the TO fields for items $[B \rightarrow cX.C_t\beta, j]$, $[C_t \rightarrow .C_t\#_t, d]$, $[C_t \rightarrow .A\gamma, e]$, $[A \rightarrow .b\beta, f]$ will be S^t .

To enrich JSR items for all the LR(1) items in I_m with FROM field, we would apply FROM procedure as $FROM(I_o'', I_m)$. Now we could see that the FROM field of JSR item of LR(1) item $[C_t \rightarrow .A\gamma, e]$ will be equal to the FROM field

of $[B \rightarrow cX.C_t\beta, j]$ which in turn is equal to the TO field of $[B \rightarrow cXC_t\beta, j]$, which is S^h . So, the set of JSR items (I_m^A) corresponding to I_m contains the following items:

$$\begin{aligned} [B \rightarrow cX.C_t\beta, j, S^h, S^t] &\in I_m^A \\ [C_t \rightarrow .C_t\#_t, d, S^h, S^t] &\in I_m^A \\ [C_t \rightarrow .A\gamma, e, S^h, S^t] &\in I_m^A \\ [A \rightarrow .b\beta, f, S^h, S^t] &\in I_m^A \end{aligned}$$

Let after performing shift operation in the state m over the symbol b , parser reaches state n . So, we must have

$$I_n = GOTO(I_m, a) = CLOSURE(\{[A \rightarrow b.\beta, g] \mid \forall g \in FIRST(\beta f)\})$$

Moreover, the TO and FROM fields of all items in the above state will be S^t . We have obtained the set of JSR items reached after performing shift over the symbol b during incremental parsing of word w_t for incremental category C_t . Also, the subtable at this state is $Tab(S^t)$.

We will now obtain the set of JSR items reached after performing shift over the symbol b during the parsing of word w for incremental category C_t . Let us consider the derivation $S \xRightarrow{*} \eta B \xRightarrow{*} \mu b \gamma \delta$ such that, B doesn't derive η . In the above derivation only one production out of $B \rightarrow cXC_t\beta$ and $B \rightarrow C_t\beta$ will be used. Hence, we have two cases of the basis of which production is used.

Case 1.1: If production $B \rightarrow cXC_t\beta$ is used, then the set of items related to state reached just after performing shift on c in the above derivation say I_x will contain the LR(1) item $[B \rightarrow cXC_t\beta, j]$ besides other items of the form $[X \rightarrow .\delta, k]$, where $k \in FIRST(C_t\beta c)$. After applying TO procedure on I_x , we should get the TO field for LR(1) item $[B \rightarrow cXC_t\beta, j]$ as S^h . Let us represent this set of items enriched with TO field as I_x'' .

After parsing X next state will be obtained by performing GOTO over I_x with symbol X . Let that state be I_y . Now we must have,

$$I_y = GOTO(I_x, X)$$

After performing GOTO we get

$$\begin{aligned} [B \rightarrow cX.C_t\beta, j] &\in I_y \\ [C_t \rightarrow .C_t\#_t, d] &\in I_y \\ [C_t \rightarrow .A\gamma, e] &\in I_y \\ [A \rightarrow .b\beta, f] &\in I_y \end{aligned}$$

After applying TO procedure over I_y , we get the TO fields of items $[B \rightarrow cX.C_t\beta, j]$ $[C_t \rightarrow .C_t\#_t, d]$ $[C_t \rightarrow .A\gamma, e]$ $[A \rightarrow .b\beta, f]$ as S^t . To enrich JSR items of LR(1) items in I_y we would apply FROM procedure as $FROM(I_x'', I_y)$. Now, we could see that the FROM field of JSR item for $[C_t \rightarrow .A\gamma, e]$ will be equal to the FROM field of JSR item for $[B \rightarrow cX.C_t\beta, j]$ which in turn is equal to the TO field of $[B \rightarrow cXC_t\beta, j]$, which is S^h . So, the set of JSR items (I_y^A) corresponding to I_y contains the following items:

$$\begin{aligned} [B \rightarrow cX.C_t\beta, j, S^h, S^t] &\in I_y^A \\ [C_t \rightarrow .C_t\#_t, d, S^h, S^t] &\in I_y^A \\ [C_t \rightarrow .A\gamma, e, S^h, S^t] &\in I_y^A \\ [A \rightarrow .b\beta, f, S^h, S^t] &\in I_y^A \end{aligned}$$

Let the state reached after performing shift of b from the state I_y be I_z . Then,

$$I_z = GOTO(I_y, b) = CLOSURE(\{[A \rightarrow b.\beta, g] \mid \forall g \in FIRST(\beta f)\})$$

Moreover, the TO and FROM fields of all JSR items of I_z will be S^t .

So, we have $I_z = I_n$. This shows that the states z and n are same. Let us name these states as q . Also, the TO fields for the items of I_z and I_n are same i.e. S^t . Hence, in this case Shift on terminal b in states m and x results only in one state q and in the sub-table for incremental category C_t .

Theorem is **proved** in this case.

Case 1.2: If $B \rightarrow cXC_t\beta$ is used. Let x is the state reached before performing the shift on b . Then, the set of items, say I_x related to state x will contain following items:

$$\begin{aligned} [B \rightarrow .C_t\beta, j] &\in I_y^A \\ [C_t \rightarrow .C_t\#_t, d] &\in I_y^A \\ [C_t \rightarrow .A\gamma, e] &\in I_y^A \\ [A \rightarrow .b\beta, f] &\in I_y^A \end{aligned}$$

After applying TO operation to I_x , the TO fields of all JSR items for above LR(1) items will be S^t .

Let I_y be the state reached after performing shift on b in the state I_x . So,

$$I_y = GOTO(I_x, b) = CLOSURE(\{[A \rightarrow b.\beta, g] \mid \forall g \in FIRST(\beta f)\})$$

Also, the TO and FROM fields of all the JSR items for the above set of LR (1) items will be S^t .

Hence, $I_y = I_n$. This shows that the states y and n are same. Let us name the state as q . Moreover, the TO fields of LR(1) items of I_y and I_n are S^t . Hence, in this case Shift on terminal b in the states m and x results only in one state q in the subtable of C_t .

Theorem is proved in this case.

As Theorem has been proved in *Case 1.1* and *Case 1.2*. So, for **Case 1** also the Theorem has been proved.

We have proved for the case containing both productions. Two other cases are when only one of these productions is present. The proof of both of these cases are very similar to the **Case 1**.

Please note that with the given set of conditions in the Theorem, we couldn't have the case in which none of the productions belong to this set.

THEOREM 1 is crucial to BPP. In the succeeding sections we will use this theorem to create our parallel parsing algorithm.

8 Block Parallelized Parser

In this section we will present our Block Parallelized Parser for LR (1) grammars. We will first give the intuition and working of BPP. Then we will present our algorithm and give a proof that our parser can accept all the LR(k) and LALR (k) languages which can be accepted by a Shift Reduce LR(k) and LALR (k) parser.

Let $G' = (N', T', P', S')$ be the augmented grammar. The incremental categories are the Non Terminals associated with the blocks to be parsed in parallel. For example, if we want to parse class definitions in parallel then we can define *class-definition* as the incremental category. Other examples can be *function-definition*, *if-statement*, *for-loop*, *while-loop* if they can be parsed in parallel. For most of the programming languages including C, C++, Java, C# above blocks can be parsed in parallel. In general we can define an incremental category to be the non terminals which derive a string containing the start of the block symbol, s_b and ending with the end of the block symbol e_b . In mathematical terms, for BPP the set of incremental category IC is defined as:

$$IC = \left\{ C_t \mid \text{if } C_t \rightarrow \alpha X \in P' \text{ then } C_t \xrightarrow{*} \alpha s_b \beta e_b \right\}$$

In the C Programming Language, a *function-definition* can be represented by the following context free grammar productions:

function-definition \rightarrow *type name* (*arguments*) *block*

block $\rightarrow s_b$ *statement*^{*} e_b

statement \rightarrow *type name* ;

statement \rightarrow *if-stmt*

statement \rightarrow *for-loop*

statement \rightarrow *while-loop*;

if-stmt \rightarrow **if** (*expression*) *block*

while-loop \rightarrow **while** (*expression*) *block*

for-loop \rightarrow **for** (*expression*; *expression*; *expression*) *block*

According to the definition of Incremental Categories above, we can see that *function-definition*, *if-stmt*, *while-loop*, *for-loop* follows the condition for an incremental category.

In a programming language there could be many types of blocks like in Java, a block related to the class definition may contain only method definitions and declaration of member variables while a block related to the method definition would contain statements including expression statements, variable declarations, if statement or loop etc. This means in a programming language not all blocks contains same type of statements. Hence, encountering the start symbol of block doesn't give us enough information about what kind of statements the block would contain. To overcome this problem we will modify the productions of incremental category such that a reduce action will happen when the start symbol is encountered. Modify the productions of $C_t \in IC$ in such a way that every production $C_t \rightarrow \alpha_t s_b X_t e_b$ is split into two productions:

$$C_t \rightarrow A_t s_b X_t e_b$$

$$A_t \rightarrow \alpha_t$$

If the productions of incremental category C_t is structured as above then during the parsing of a word related to this incremental category there will be reduce action to reduce the current symbols to production $A_t \rightarrow \alpha_t$ when s_b becomes the look ahead symbol. As each A_t is related to only one C_t and vice-versa, we can easily determine which incremental category has to be parsed next.

Now we are in a stage to define Block Parallelized Grammar. Given a grammar $G = (N, T, P, S)$ and a set of incremental categories

$$IC = \left\{ C_t \mid C_t \in \alpha s_b X e_b \in P' \text{ and } C_t \xrightarrow{*} \alpha s_b \beta e_b \right\}$$

we define Block Parallelized Grammar $G^P = (N^P, T, P^P, S)$ such that

$$P^P = P \cup \{ C_t \rightarrow A_t s_b X_t e_b, A_t \rightarrow \alpha_t \mid C_t \rightarrow \alpha_t X_t \in P \} - \\ \{ C_t \rightarrow \alpha_t X_t \mid C_t \rightarrow \alpha_t X_t \in P \}$$

$$N^P = \{ A_t \mid C_t \rightarrow A_t s_b X_t e_b, A_t \rightarrow \alpha_t \forall C_t \rightarrow \alpha_t X_t \in P \}$$

and N^P is partitioned using FIRSTNT as given in Section 7.

Now we can use THEOREM 1 to create BPP. Let us have a string $w = \omega a \delta s_b \eta e_b \mu$ where $\omega, \delta, \eta, \mu \in T^*$, $A_t \xrightarrow{*} a \delta$ and $X_t \xrightarrow{*} s_b \eta e_b$. During the parsing w , when s_b is encountered we should have a reduce action to A_t , based on the production $A_t \rightarrow \alpha_t$. Now, we can get C_t associated with A_t . According to THEOREM 1, during parsing of the word w if the parser reaches at state q in the sub table of C_t after performing shift action on a then during the incremental parsing of the word $w_t = \omega a \delta s_b \eta e_b$ for the incremental category C_t the parser will also reach the state q in the sub table of C_t after performing the shift action on a . This means, we can replace the state reached just before performing shift action on a with the start state of subtable of C_t and w_t can now be parsed incrementally.

It is now evident that why the partitions of Non Terminals should be created as described in Section 7. If the partitions are not created in this way, then it may happen after a shift on a during the parsing of w and incremental parsing of w_t may reach the same state but not in the same sub-table. By creating partitions as described in Section 8, we make sure that when s_b is encountered by BPP then the newly created parallel parser knows in which sub-table it has to continue the parsing in. On the other hand if partitions are not created as described in Section 8, then newly created parallel parser wouldnt know in which sub-table it has to continue parsing in.

This property is used by BPP to parse the block incrementally. Algorithm 1 is the BPP parsing algorithm of incremental table S^t . If $t = 1$, then the algorithm is for the Top Level Block. Otherwise it is for other incremental category.

This parsing algorithm is for any block be it top level block or child block. Lines 1-4 initializes different local variables. Lines 5-32 is the main loop of algorithm which does the whole work. Line 6, gets the top state on the stack. Lines 7-9 pushes the next state on the stack if there is a shift operation. Similarly, lines 10-11 changes current table if there is a jump operation. Line 12-27 are executed if there is reduce action which reduces according to the production $A \rightarrow \beta$. Line 14 checks if current input symbol is a start of the block symbol and if reduce action reduces $A_t \rightarrow \alpha_t$ for an incremental category C_t . If yes then Lines 15-22 gets C_t related to A_t , pops $|\alpha_t|$ states from stack and pushes these states and start state to a new stack, creates and starts a new BPP for C_t and shifts to the end of block. In this case next symbol will become C_t . If check of Line 14 fails then it means this is a regular reduce action not associated with any block. Lines 24-27, pops $|\beta|$ states from stack and shifts to a new state. Line 28 returns if there is an accept action. Accept action can be for both Top Level block and Child Block. Line 30 reports error if none of the above cases are satisfied.

Fig. 2 shows an example of how BPP works. It will start parsing the block

Algorithm 1 Block Parallelized Parsing Algorithm

```
1:  $a = \text{start symbol of input}$ 
2:  $\$ = \text{last symbol of input}$ 
3:  $h = \text{initial parsing subtable}$ 
4:  $\text{stack} = \text{stack of states}$ 
5: while true do
6:    $s = \text{stack.top}()$ 
7:   if  $\text{Tab}(S^h)[s, a] == \text{shift } t$  then
8:      $\text{stack.push}(t)$ 
9:      $a = \text{next symbol of input}$ 
10:  else if  $\text{Tab}(S^h)[s, a] == \text{jump } k$  then
11:     $h = k$ 
12:  else if  $\text{Tab}(S^h)[s, a] == \text{reduce } A \rightarrow \beta$  then
13:     $b = a$ 
14:    if  $a == s_b$  and  $\text{Tab}(S^h)[s, a] == \text{reduce } A_t \rightarrow \alpha_t$  then
15:       $\text{get } C_t \text{ related to } A_t$ 
16:       $\text{stack}_t = \text{new stack of states}$ 
17:       $\text{stack}_t.\text{push}(\text{start state of } C_t)$ 
18:       $\text{pop } |\alpha_t| \text{ states from } \text{stack}_t \text{ and push them to } \text{stack}_t$ 
19:       $\text{create new block parser related to } C_t \text{ with } \text{stack}_t$ 
20:       $\text{start new block parser}$ 
21:       $\text{go to the end of this block}$ 
22:       $a = C_t$ 
23:    else
24:       $\text{pop } |\beta| \text{ states from stack}$ 
25:       $t = \text{stack.top}()$ 
26:      if  $\text{Tab}(S^h)[t, A] == \text{shift } p$  then
27:         $\text{stack.push}(p)$ 
28:    else if  $\text{Tab}(S^h)[s, a] == \text{accept}$  then return
29:    else
30:       $\text{error}$ 
```

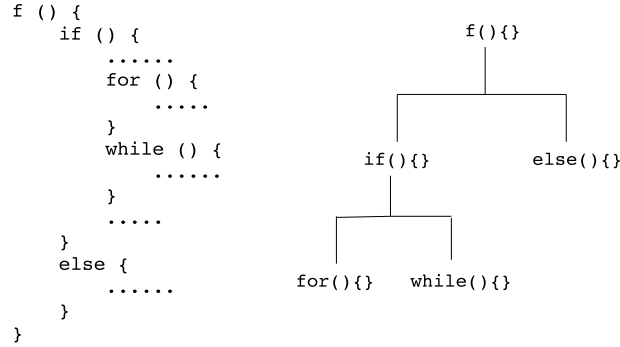


Fig. 2. Example of BPP parsing source code

of function f . When it will encounter an *if* block a new BPP in another thread will be created which will parse *if* block. Parent BPP will move ahead to the end of *if* block and will also create another thread to parse *else* block. In this way input is divided into different threads parsing each block.

In this algorithm we have tried to minimize the amount of serial work to be done to get to the end of the block for the new block parser. One BPP doesn't have to do any communication with other BPPs. Also, there are no side effects of the above algorithm. All the variables which are being modified are local variables. Hence, there is no need of synchronization. This also reduces any amount of cache contention between different processors. Generation of Abstract Syntax Tree or Parsing Tree is easy using above algorithm and it requires very little change in the above algorithm.

It may be argued that the step go to the end of this block is a serial bottleneck for the parallel algorithm. [15] describes an algorithm to perform lexical analysis of string in $O(\log n)$ time using $O(n)$ processors in a parallel fashion. When performing lexical analysis in parallel as described in [15], lexer could store the start symbols of a block with its corresponding end symbol for that block. Now determining the end of block is just a matter of searching through the data structure. Many ways exist to make this searching as fast as possible like using a Binary Search Tree or a Hash Table.

9 Comparison with other Parallel Parsing Algorithms

In this section we will show how our technique is better than other techniques. [14] developed a technique which divides whole grammar into n sub-grammars which are individually handled by n sub-compilers. Each sub-compiler needs its own scanner which can scan a sub-grammar. It requires an automatic tool to generate sub-compiler. This technique requires significant changes in not only in Parser and Grammar but also in Lexical Analyzer phase. Contrast to this our Block Parallelized Parser is easy to generate as our technique does not require

any change in the grammar and lexical analyzer and it is pretty easy to modify current YACC and Bison Parser Generator tools to support the generation of our parser.

LR substring parsing technique described in [2] is specifically for Bounded Context (1, 1) grammars. There are no limitations like this to Block Parallelized Parser. Although in this paper we have shown how we can create an LR (1) Block Parallelized Parser but we believe it can be extended to LR (k) class of languages and also could be used by LALR (1) parser. Hence, our technique accepts a larger class of grammars.

[8], [7], [9], [11] all develops algorithm for parsing LR (k) class of languages in parallel. These techniques and in all other techniques the creation of Abstract Syntax Tree is not as easy as it is in our technique. Moreover our technique is simpler than all others.

Hence, we could see that Block Parallelized Parser is easy to construct, accepts wider class of languages and supports an easy construction of Abstract Syntax Tree.

10 Implementation and Evaluation

We implemented Lexer, Block Parallelized Parser and Shift Reduce LR (1) parser for C Programming Language supporting a few GNU C extensions required for our tests. Implementation was done in C# Programming Language. To simplify our implementation we only included function-definition as the Incremental Category for BPP. Moreover, function-definition would still give us a sufficient amount of parallelism as we would see in the evaluation. We modified the Lexer phase so that it will keep track of the position of s b and its corresponding e b. This information was stored in the form of a C# Dictionary (which is implemented as a Hash Table) with the position of s b as the key and position of e b as the value. As, thread creation has significant overhead so we used C# TaskParallel Library which is Thread Pool implementation in C#. Our implementation doesn't have a C preprocessor implementation. So, we first used gcc to perform preprocessing and the preprocessed file is used as input to our implementation.

We evaluated the performance of BPP with Shift Reduce LR (1) parser by parsing 10 random files from the Linux Kernel source code. We compiled C# implementation using Mono C# Compiler 3.12.1 and executed the implementation using Mono JIT Compiler 3.12.1 on machine running Fedora 21 with Linux Kernel 3.19.3 with 6 GB RAM and Intel Core i7-3610 CPU with HyperThreading enabled.

In C Programming Language, preprocessing `#include` could actually generate very long files. Normally, the header files contains declarations not function definitions. So, this leads to less amount of parallelism being available. Hence we decided to evaluate with including header files and excluding header files. Fig. 3 shows the performance improvement with respect to Shift Reduce LR(1) parser of 10 random Linux Kernel files. Fig. 3 shows performance improvement for both cases including the header files and excluding the header files. As expected we

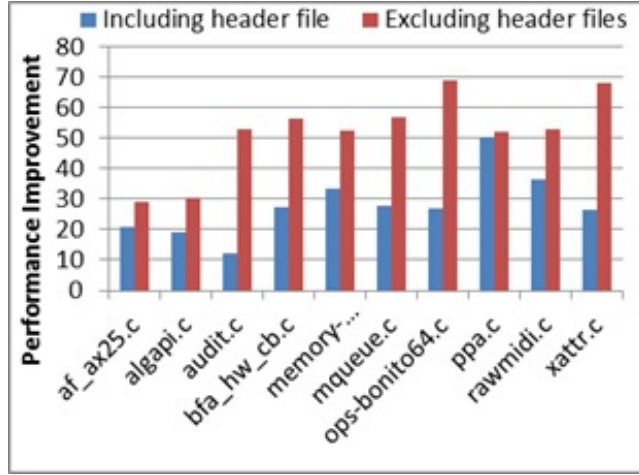


Fig. 3. Results of parsing 10 Linux Kernel Source Files.

could see that the performance improvement with excluding the header files is more than the performance improvement including the header files.

The performance improvement in the case of excluding header files matters most for the programming languages like Java, Python, C# where most of the program is organized into blocks the results because in these programs the amount of parallelism available is high.

The average performance improvement in the case of excluding header files is 52% and including header files is 28%.

11 Conclusion

In this document we present our Block Parallelized Parser technique which could parse the source code in a parallel fashion. Our approach is a divide and conquer approach in which, the divide step divides the source code into different blocks and parse them in parallel whereas the conquer step only waits for all the parallel parsers to complete their operation. It is based on the Incremental Jump Shift Reduce Parser technique developed by [13]. Our technique doesn't require any communication in between different threads and doesn't modify any global data. Hence, this technique is free of thread synchronization. We develop this technique for LR (1) languages and we believe that it can be extended to accept LR (k) languages and could be converted to an LALR (1) parser easily. Our technique doesn't do any major changes in the parsing algorithm of a Shift Reduce Parser hence the Abstract Syntax Tree can be created in the same way as it has been creating in Shift Reduce Parser. Moreover, our parser can also work as an Incremental Block Parallelized Parser. We implemented Block Parallelized Parser and Shift Reduce LR (1) Parser for C Programming Language in C#. The performance evaluation of BPP with Shift Reduce LR (1) parser

was done by parsing 10 random files from the Linux Kernel source code. We compiled C# implementation using Mono C# Compiler 3.12.1 and executed the implementation using Mono JIT Compiler 3.12.1 on machine running Fedora 21 with Linux Kernel 3.19.3 with 6 GB RAM and Intel Core i7-3610 CPU with HyperThreading enabled. We found out that our technique showed 28% performance improvement in the case of including header files and 52% performance improvement in the case of excluding header files.

12 Future Work

Our parser accepts LR (1) languages we would like to extend it to accept LR (k) languages. In our technique, the parser determines when to create a new parallel parser thread. If the responsibility of this decision can be given to the lexical analysis phase then the lexical analysis can actually start the parsers in parallel. This will lead to significant performance advantage. Moreover, our technique has been applied to languages which doesn't have indentation in their syntax like the way Python has. [24] shows an efficient way to parse the language which has indentation as a mean to determine blocks. Our parser can be extended to accept those languages also. We are working towards developing a Block Parallelized Compiler which could compile different blocks of a language in parallel. Block Parallelized Parser is one of the components of a Block Parallelized Compiler. Semantic Analysis phase also share the same properties as the Syntax Analysis phase. In Programming Languages, an entity like variable or type is declared before using it. So, in this case also a lot can be done to actually parallelize the semantic analysis phase.

References

1. Neal M. Gafter and Thomas J. LeBlanc. Parallel Incremental Compilation. Department of Computer Science, University of Rochester: Ph. D. Thesis. 113
2. Gwen Clarke and David T. Barnard. 1996. An LR Substring Parser Applied in a Parallel Environment, Journal of Parallel and Distributed Computing.
3. G. V. Cormack. 1989. An LR Substring Parser for noncorrecting syntax error recovery, ACM SIGPLAN Notices
4. Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffery D. Ullman. 2007. Compilers Principle Tools and Techniques Second Edition, Prentice Hall
5. Floyd, R. W. Bounded Context Syntactic Analysis. Comm. ACM 7, 21 February 1961
6. J. H. Williams, 1975. Bounded context parsable grammars. Information and Control 28, 314-334
7. R. M. Schell, 1979. Jr. Methods for constructing parallel compilers for use in a multiprocessor environment. Ph. D. thesis, Univ. of Illinois at Urbana-Champaign
8. J. Cohen and S. Kolodner. 1985. Estimating the speedup in parallel parsing, IEEE Trans. Software Engrg. SE-11
9. C. N. Fischer, 1975. On parsing context free languages in parallel environments. Ph. D. thesis, Cornell Univ.

10. M. D. Mickunas and R. M. Schell. 1978. Parallel Compilation in a multi-processor environment (extended abstract). J. Assoc. Comput
11. D. Ligett, G. McCluskey, and W. M. McKeeman, Parallel LR Parsing Tech. Rep., Wang Institute of Graduate Studies, July 1982
12. Edward D. Willink, Meta-Compilation for C++, University of Surrey, June 2001
13. Pierpaolo Degano, Stefano Mannucci and Bruno Mojana, Efficient Incremental LR Parsing for Syntax-Directed Editors, ACM Transactions on Programming Languages and Systems, Vol. 10, No. 3, July 1988, Pages 345-373.
14. Sanjay Khanna, Arif Ghafoor and Amrit Goel, A PARALLEL COMPILATION TECHNIQUE BASED ON GRAMMAR PARTITIONING, 1990 ACM 089791-348-5/90/0002/0385
15. W. Daniel Hillis and Guy L. Steele, DATA PARALLEL ALGORITHMS, Communications of the ACM, 1986
16. Bison, <http://www.gnu.org/software/bison>
17. YACC, <http://dinosaur.compilertools.net/yacc>
18. Clang, <http://clang.llvm.org>
19. Mono, <http://mono-project.com>
20. El-Essouki, W. Huen, and M. Evans, Towards a Partitioning Compiler for Distributed Computing System, IEEE First International Conference on Distributed Computing Systems, October 1979.
21. Thomas J. Pennello and Frank DeRemer, A Forward Move Algorithm for LR Error Recovery, Fifth Annual ACM Symposium on Principles of Programming Language.
22. Jean-Philippe Bernardy and Koen Claessen, Efficient Divide-and-Conquer Parsing of Practical Context-Free Languages.
23. L. Valiant. General context-free recognition in less than cubic time. J. of Computer and System Sciences, 10(2):308-314, 1975.
24. Micheal D. Adams. Principled Parsing for Indentation-Sensitive Languages, ACM Symposium on Principles of Programming Languages 2013. January 23-25.